Lingling Jacobson

Comp 524

Assignment 4

Lazy Evaluation

Lazy evaluation is for lazy coders…but not really. Lazy evaluation describes the strategy of delaying the evaluation of an expression until its value is needed. Lazy evaluation can reduce time complexities and can discard unneeded sub-expression. It is a useful tool that many software engineers should have in their toolbelt. However, it is important to know when, why, and how to use lazy evaluation. Used incorrectly or at the wrong time, it might hurt the program instead of help it.

A simple example of this is: a = 1 + 2. The expression 1 + 2 is not actually evaluated until the variable ‘a’ is used. With this example, the variable a may never be used, and the compiler saves time complexities by not evaluating ‘1 +2’. It might be hard to understand lazy evaluation at first, because why would a coder write unused code? Why would someone define variable a if they were never going to use it? The above example is not a common example of the use of lazy evaluation, rather it is just the simplest to explain. One of the most common times lazy evaluation is used is with conditionals. The if statement: if( 0 > 3 && 1 < 4) would stop after 0 > 3 if lazy evaluation is used because we know the if statement will be false no matter what 1 < 4 evaluates to. It is more common for a real program to have variables in the conditionals, because there would be no use to hard code an if statement to always be false. With changing variables, the if statement evaluations could change, and having lazy evaluations will disregard the second expression if the first is false.

Lazy evaluation works by using memoization. Memoization is more commonly used with recursion, however, it is still used during lazy evaluation. Memoization can speed up programs, by acting like a dictionary and storing the result of the evaluation of smaller evaluations. It is used for dynamic programming where the final result is evaluated by other smaller expression evaluations. Additionally, it is good for expensive programs that have repeated parts or evaluations. Therefore, when a program has lazy evaluation, the instance the program needs the expression to be evaluated, it can look it up using memoization.

Something very useful and one of the best reasons to use lazy evaluation is that a program can create an infinite list using lazy evaluation. Normally, a compiler will crash if the program has an infinite list or an infinite list. Lazy evaluation can create an infinite list, because it won’t evaluate it for infinity. It will only evaluate the parts of the list that are used. A good example is creating an infinite list of all integers (a loop that adds a new integer to the list that is +1 of the previously added integer). Sometimes a program might need to access a range of 10 numbers, but we can’t anticipate which 10 numbers it will need. Therefore, the program needs to have access to any 10 integers, so an infinite list of all integers is a great way to achieve this. If the range was 0-9, lazy evaluation will only evaluate run through the loop and evaluate the first 10 run throughs. If the range was 10,000-10,009, the program would only evaluate the 10,000th-10,009th iterations of the loop. No matter what range is selected, if it is still 10 numbers, the program will only evaluate those 10 numbers. This is how the compiler doesn’t crash and throw an error.

One of the disadvantages of lazy evaluation is that if the program reuses the variable that has an expression, it has to reevaluate it. Lazy evaluation’s main purpose is to not spend time evaluating unneeded expressions, so having to evaluate the same expressions multiple times is almost counterproductive. A coder must know when and when not to use lazy evaluation in order for it to be most effective. It just requires people to be more careful and take their time when they write their program.

Lazy evaluation can be used in racket in the same way. First, just set the #lang in racket to ‘lazy’. Lazy should be used for conditionals, racket/list libraries, and racket/base libraries. In racket, we can use both lazy and strict evaluation. Therefore, we can use lazy where it’s better to use lazy and strict where it’s better to use strict. Strict evaluation is better used for I/O mutations, parameterization, etc. There are many functions and lazy variants of functions in racket; lazy evaluation is very compatible with racket.

A lazy student might not sound like the best student, but sometimes it pays to be lazy. Why spend time and energy evaluating expressions that don’t actually need to be evaluated? Lazy evaluation is a strategy that all software engineers should know about. The ability to make and utilize an infinite list is a useful skill that we should all know how to do. Lazy evaluations uses memoization, which is a sort of dictionary to look up the expressions, and once the expression is reached to a point where it is needed, then it will be looked up using a key with memoization. However, since memoization saves the expression, every time the expression is needed, it would have to re-evaluate. Therefore, it would end up taking more time to re-evaluate the expression every time. So there are times to use lazy evaluation and strict evaluation. In Racket, we are able to use both interchangeably, which allows us a lot of flexibility in writing our code. This coding strategy saves time and is more efficient. It is important for coders to know how it works so that they can also use it.

Sources

<https://www.tutorialspoint.com/functional_programming/functional_programming_lazy_evaluation.htm>

<https://docs.racket-lang.org/lazy/>

<https://www.interviewcake.com/concept/java/memoization> <https://www.cloudsavvyit.com/12446/what-is-memoization-and-why-does-it-matter/>

<https://medium.com/background-thread/what-is-lazy-evaluation-programming-word-of-the-day-8a6f4410053f>